6.005 Elements of Software Construction
Fall 2008

6.005 elements of software construction

designing state machines

Daniel Jackson

Until now, we've been doing a crash course in Java. This is the first lecture of the course proper, starting with state machines -- our first software paradigm.

a lesson from failure

friendly fire in afghanistan

Photograph of PLGR removed due to copyright restrictions.

Afghanistan, December 2001

- US soldier uses plugger* to mark Taliban position for air-strike
- notices battery-low warning, so replaces battery and calls in coordinates
- resulting strike kills user and two comrades and wounds 20 others

what happened?

- replacing battery reset to current position

*PLGR: precision lightweight global-positioning satellite receiver

Vernon Loeb. Friendly Fire Deaths Traced to Dead Battery; Taliban Targeted, but U.S. Forces Killed. *Washington Post*. March 24, 2002

A tragic story reported in the Washington Post.

what can we learn from this?

accidents are complex

- rarely one cause, so be wary of simple explanations
- often human factors + technology

lessons from this case

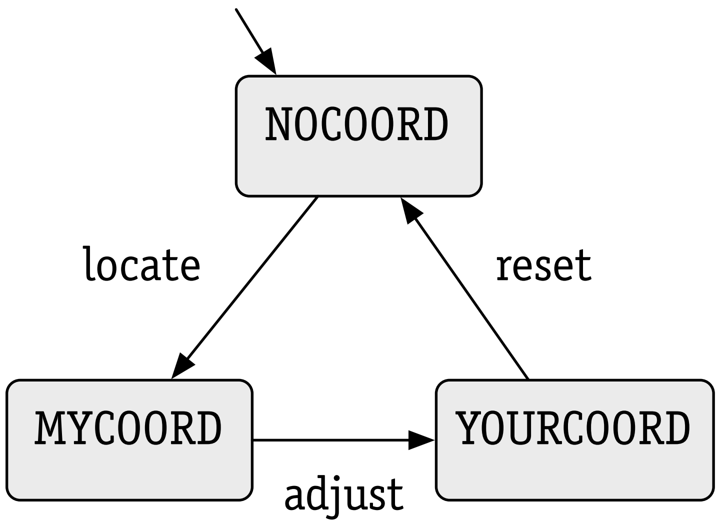
- design for robustness against all likely failure modes
- defaults are dangerous: user should be warned
- <u>describe and analyze all usage scenarios</u>

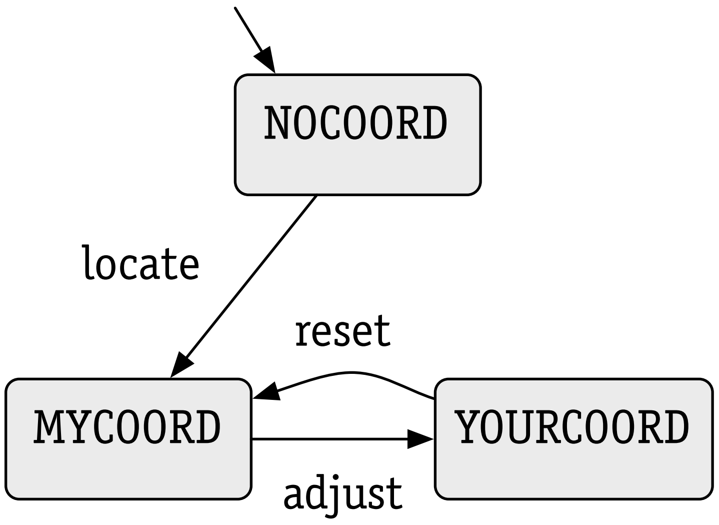
There are lots of problems here that could be analyzed. For example, there's clearly a user interface problem that the device didn't clearly indicate when it was showing calculated coordinates versus the coordinates of the user. Like many accidents, this one involved human factors issues in which the designers didn't think through all the ways that user might interact with the machine. But whatever the problem was, it's clear that we need some way to talk about the possible scenarios in simple but precise way.

state machines

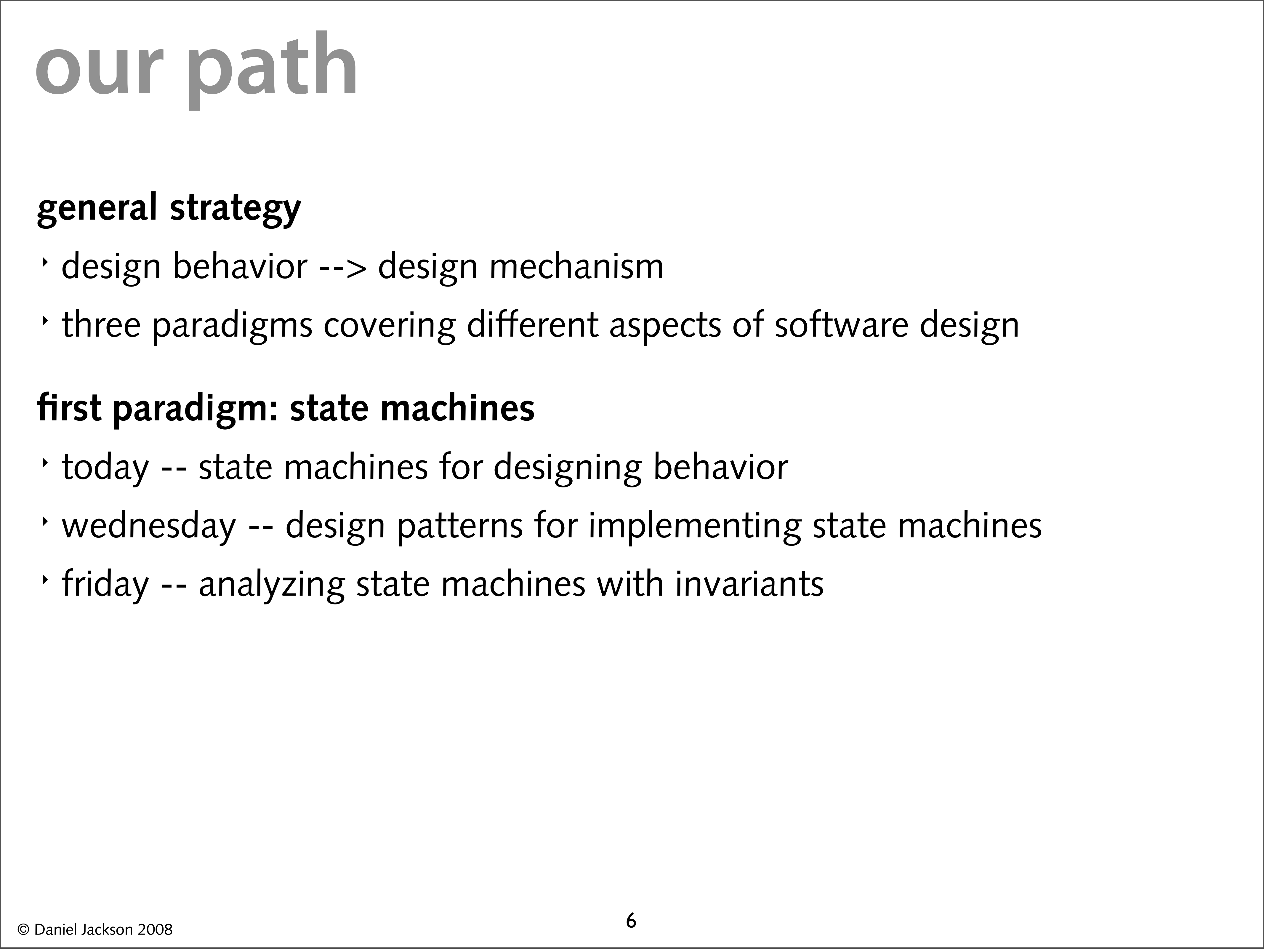
what they are

- a simple notation for describing behaviors
- succinct and abstract -- but not vague!
- basis for analysis and for implementation





State machines give us this simple and precise way of talking about scenarios. Don't confuse abstract with vague! They're abstract in the sense that they allow you to express only the important bits, but that isn't the same as being vague. A state machine doesn't tell you everything there is to know about a system, but what it does tell you -- namely what orders events can happen in -- it tells you very precisely.



our path

general strategy

- design behavior --> design mechanism
- three paradigms covering different aspects of software design

first paradigm: state machines

- today -- state machines for designing behavior
- wednesday -- design patterns for implementing state machines
- friday -- analyzing state machines with invariants

This strategy is one we'll follow throughout the course: first learning how to think about designing the behavior of a system, and only then thinking about designing the mechanism -- the system's internal structure. This is a very important 'separation of concerns'. It lets you focus on the behavior without getting mixed up with mechanism issues. And there's an important sequencing here too: how can you figure out what mechanism you need to implement a behavior if you don't know what the behavior is? For small systems, many programmers just do the behavioral design completely in their heads, but that can lead you into real trouble -- and many late nights of restructuring and debugging. For large systems, teams usually develop requirements and specification documents. What we're trying to teach you is the essential notions you need in those documents. The models we're teaching you in this course are much more succinct than informal documentation, can be analyzed automatically by tools (although that's beyond this course) and can be translated fairly directly into code, as we'll see. So constructing these behavioral designs makes it easier to write the code, because they help bridge the gap -- the code doesn't come out of thin air.

designing a midi piano

a midi-piano

functionality required

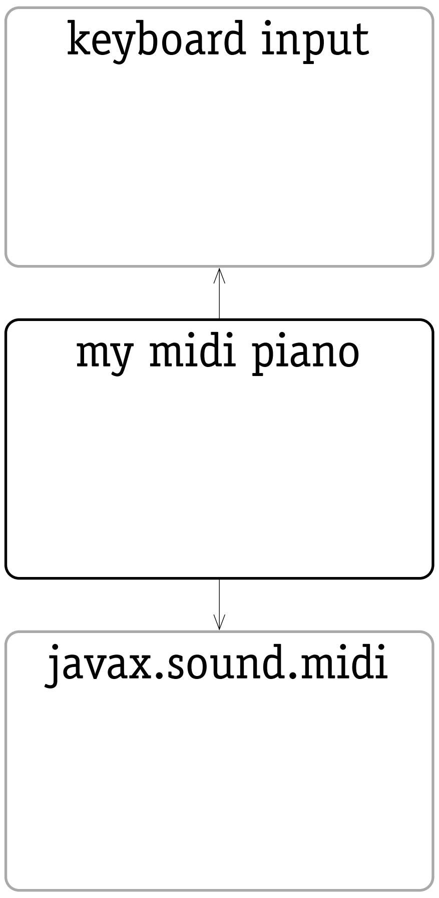
- play notes on computer keyboard
- sustain by holding key down
- cycling through instruments
- record and playback

context

- piano depends on keyboard, midi API
- this is a dependence diagram -- more later

need to start by understanding

- keyboard input: press and release
- MIDI interface: commands



We'll see more of these kinds of diagrams later. They're called _dependency diagrams_. Note that the edges aren't labelled: that's a clue that they're not state machines! Sometimes people try and make these various notations look different from one another (eg, by making the boxes slightly different shapes), but our experience is that it's more trouble than it's worth: easier just to use plain old boxes and arrows and label your diagrams so you know what they are.

Note, btw, that there's a big decision that's already been made here: what the context actually is. We could have chosen to use a different API for making the musical sounds, and we could have implemented our own keyboard driver. Just defining where your system sits with respect to the outside world and your assumptions about it is a big step.

modeling the context

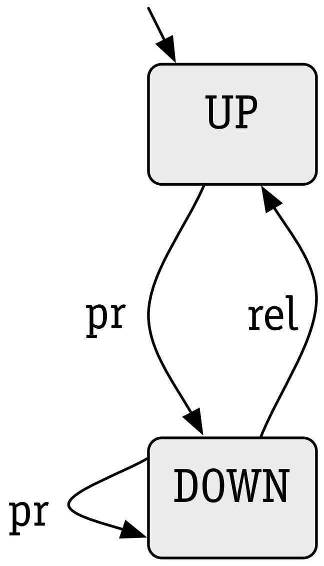
a single key

the state machine

- two states, UP and DOWN
- UP is the initial state
- two input event classes, with these designations:

  pr: the keyboard driver reports a key press

  rel: the keyboard driver reports a release



meaning: a set of traces

- a trace is a sequence of events
- traces of this machine are

  <>

  <pr>

  <pr, rel>

  <pr, pr, rel>

  ...

The designations are more important than you might think. The biggest mistake novices make with state machine modeling is that the event classes aren't right. Formulating a designation -- trying to say exactly what comprises the occurrence of an event -- is a big help in checking whether your notion of the event class is reasonable, and in conveying it to others. Here the crucial idea is that pr, for example, doesn't mean that the user actually pressed the key, which is an important distinction.

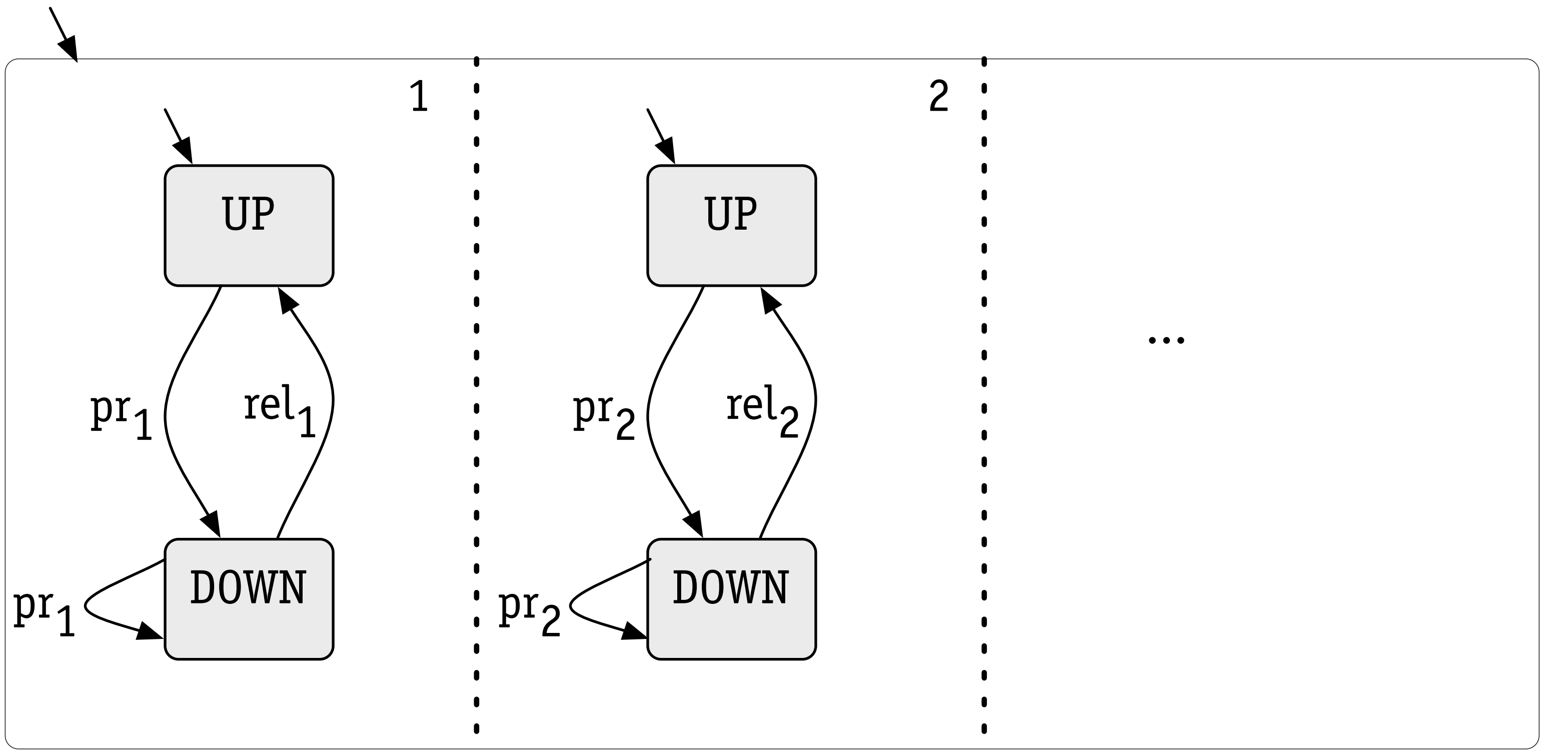
Interestingly, not all keyboard drivers generate key repeats in this form. In Gnome, for example (as a student told me after class), the driver actually repeats release-press pairs -- in the notation we'll see next week, the repeat is (rel pr)* rather than pr*. That's bad news, because it's not at all clear how the generated repeats can be distinguished from the real key presses.

Note that the trace set is infinite if the machine has a loop in it, although each trace is itself finite -- the history of events up to some point in time. This notion of traces was invented by Tony Hoare in a machine formalism called Communicating Sequential Processes. You may be a bit confused if you've seen the notion of languages and state machines in a computer science theory setting. There, the language usually consists of the set of all complete sequences, which are defined because there are some special states that are marked as final. Here we only have initial states.

the whole keyboard

represent as parallel combination of state machines

- each key's machine can take steps independently
  (general rule: shared events must be synchronized, but no sharing here)
- traces include <>, <pr1>, <pr1, rel1>, <pr1, rel1, pr2>, <pr1, pr2, pr1>

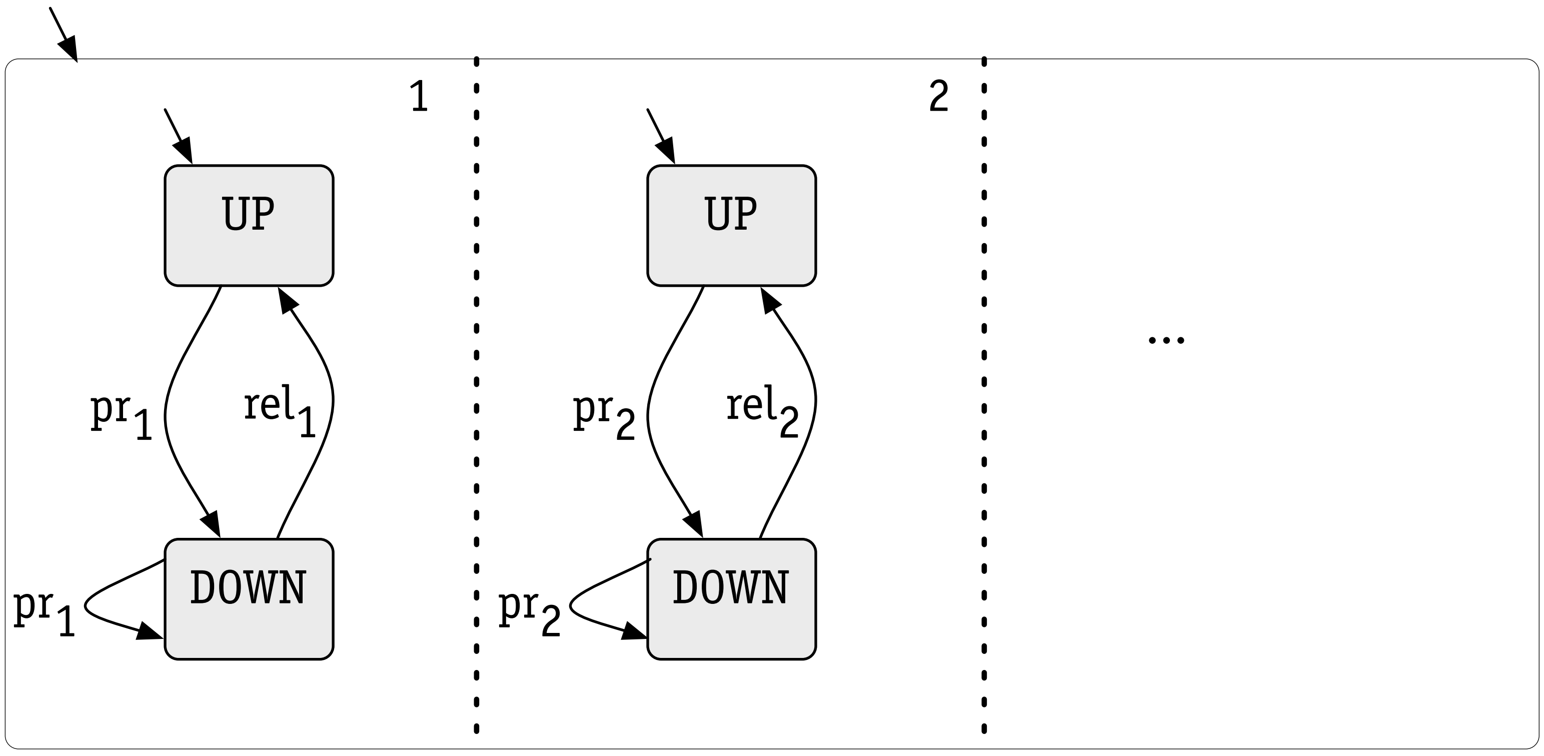


The parallelism here is just conceptual. To implement this kind of parallelism, you could actually have threads running concurrently, but usually you just have separate state components, one for each of the submachines, and the whole machine runs sequentially with one of the submachines taking a step at a time. When the ‘alphabets’ of events of the component submachines overlap, the shared events have to happen as one. In this case, there aren’t any shared events, but we’ll see examples of this below.

approximate models

over-approximation

- this model is actually an over-approximation
- it allows more traces than can happen (eg <pr1, pr2, pr1>) -- try it and see
- but this is OK: problem is handling fewer inputs than can occur



This is an overapproximation because it includes traces that can't happen in practice. In a real keyboard driver, the keys are not independent. In many, for example, the key repeat on one key is terminated when another key is pressed, so the trace <pr1, pr2, pr1> can't occur, since the pressing of key 2 prevents repeats of key 1.

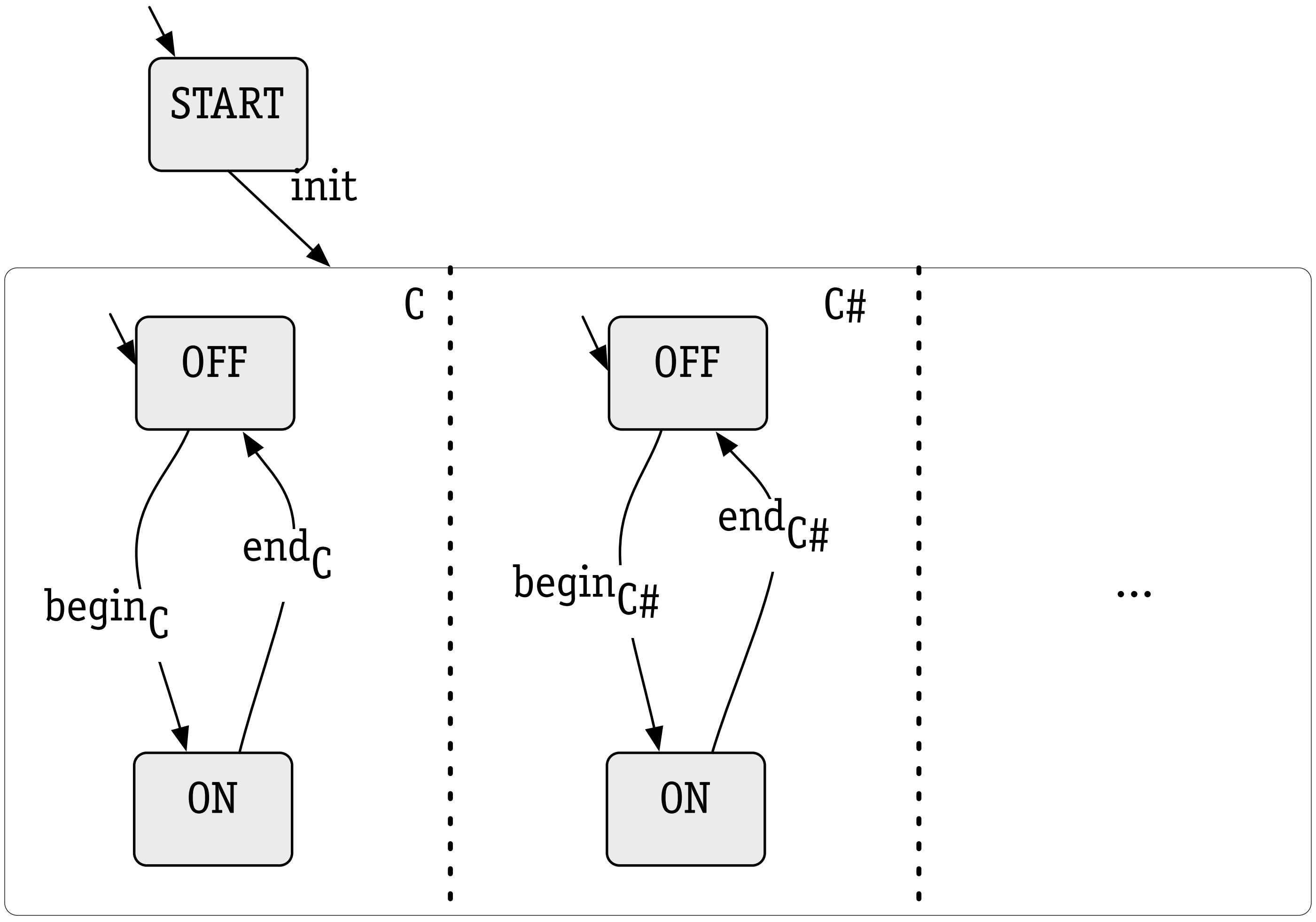
Overapproximation is good because it allows us to use a simpler model. It doesn't matter if we handle more input scenarios than actually happen.

[Note, not explained in lecture: There's actually a form of overapproximation for output too, called nondeterminism, in which you show multiple possible outputs happening with the implication that either they are all equally acceptable, or that some other part of the specification which is not currently being considered will say which should happen. This is useful because, again, it makes the model simpler. But there's a more subtle reason it helps: if I can show that my system has the right effect whichever output behavior it chooses, then I've shown that any of the implementations you choose will be OK. Sometimes we just want to make sure that everything will be OK without specifying exactly what the behavior will be. An example of this is giving a caching algorithm without saying exactly what the rules are for choosing which line to evict. Sometimes we actually can't control the behavior. In a distributed system, for example, it's often impossible to get events at different hosts to execute in some given ordering, so it's better to allow whatever ordering can happen, and make sure the system will work whichever it is.]

modelling the midi API

simple interface

- just turn each note on and off; need explicit initialization



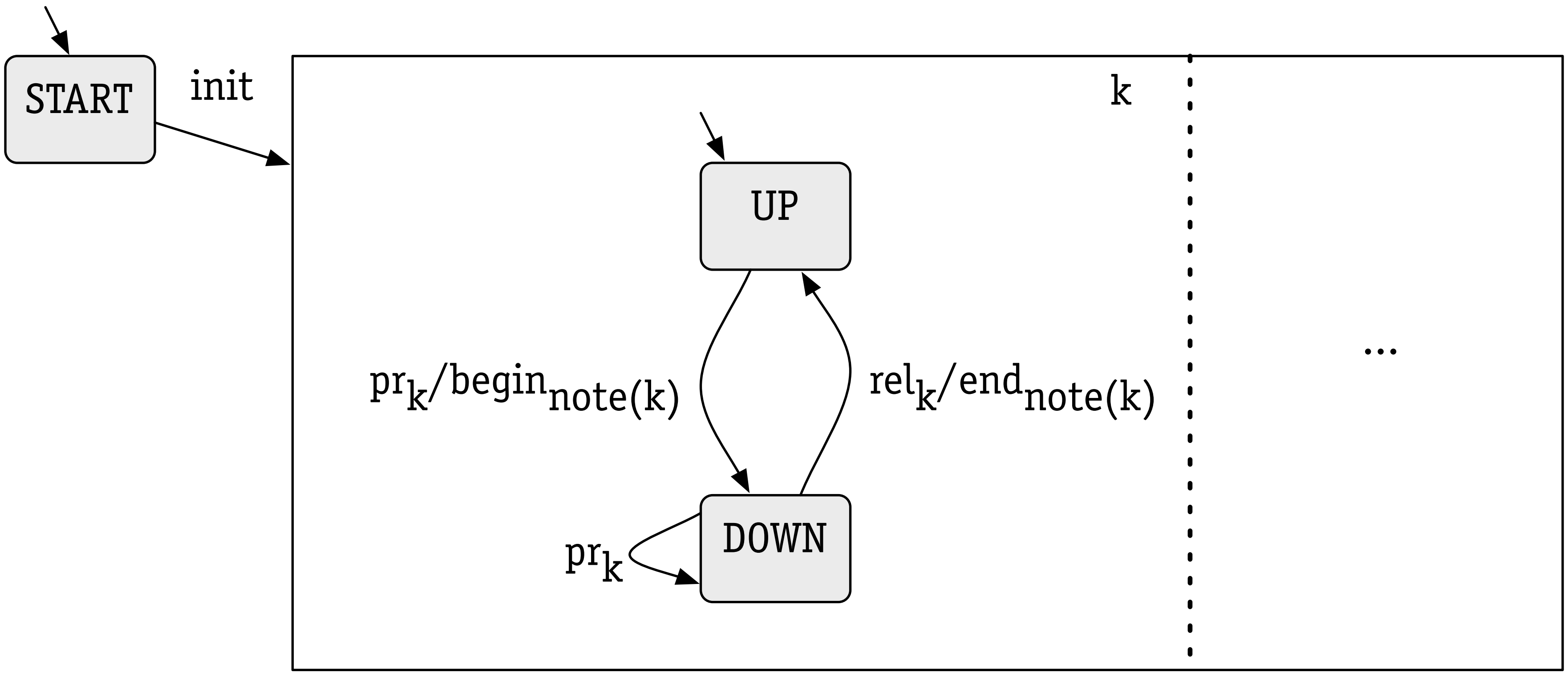
There’s nothing much going on here: just a simple state machine protocol, just like the kind many APIs present. File i/o, for example, has the same kind of protocol: you can’t read or write when the file is closed, etc.

first design draft

first design draft

design strategy

- base structure on input and embed output events
- absorb superfluous presses



This shows the result of starting with the basic keyboard and adding output events. I've written this as a generic machine, with the understanding that there is one of these for each key. I've assumed some function, note, that takes a key and returns a musical note associated with it. So begin_note (k) means the begin event for the note associated with key k. Note that there's no output generated for the repeated press events.

In lecture, I referred to the filtering of the superfluous presses as 'debouncing', but that's an abuse of terminology. Debouncing means eliminating the spurious press and release events associated with the electrical contact of the key's switch not closing cleanly.

state machine semantics

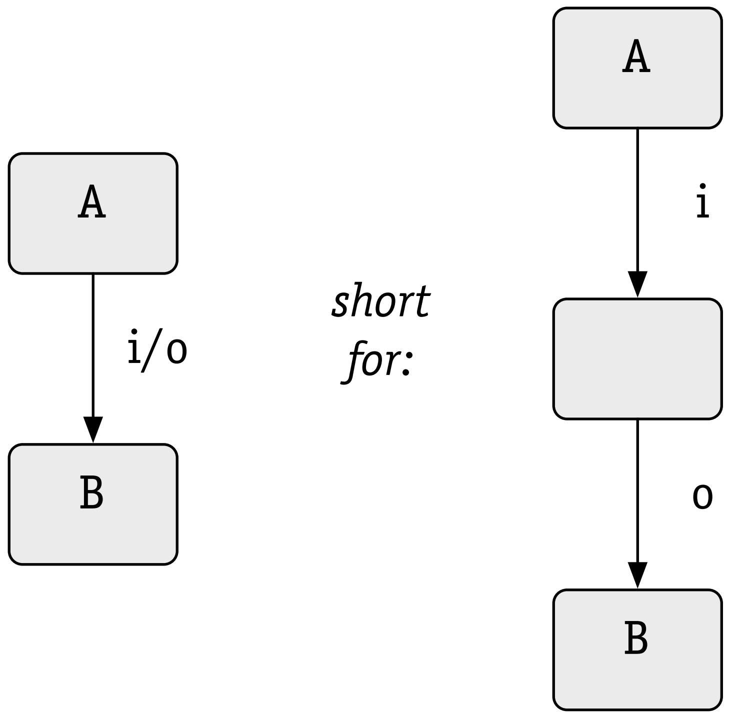
the i/o shorthand

input/output shorthand

- i/o label is just a shorthand

subtle consequence

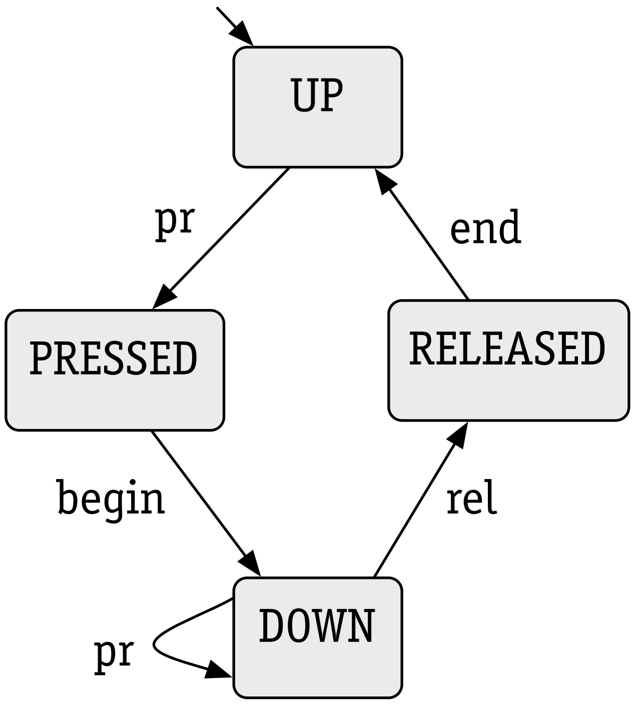
- o need not follow i immediately
- another event can intervene



© Daniel Jackson 2008

There are different ways that a state machine formalism can handle inputs and outputs. This is perhaps the simplest way. We just say that after the input occurs, the output can happen -- and no other inputs until it’s happened. Note that we don’t actually distinguish inputs from outputs in any formal way, although of course when you designate the events you’ll say which are which. And we don’t have any notion of outputs following inputs immediately: if this was a component submachine in a parallel combination, another machine could take a step between the i and the o, so that the o doesn’t immediately follow the i. In most situations this is exactly what we want anyway.

a state machine is ...



- a set of states

  State = { UP, DOWN, PRESSED, RELEASED }

- a set of initial states

  Init = { UP }

- a set of events

  Event = { pr, rel, begin, end }

- a transition relation

  trans ⊆ State × Event × State =

  { (UP, pr, PRESSED), (PRESSED, begin, DOWN), (DOWN, pr, DOWN), (DOWN, rel, RELEASED), (RELEASED, end, UP) }

- trace set (derived from trans and Init)

  traces = { <>, <pr>, <pr, begin>, <pr, begin, pr>, <pr, begin, rel>, ...}

Here's the mathematical definition of a state machine that we're using. The diagram can be translated easily into this structure: each box becomes a member of the set of states, each label name becomes an event class, each edge becomes a tuple in the transition relation. In terms of what the machine actually means though we only care about the traces: the states are just an artifact to help us define the order in which the events can occur.

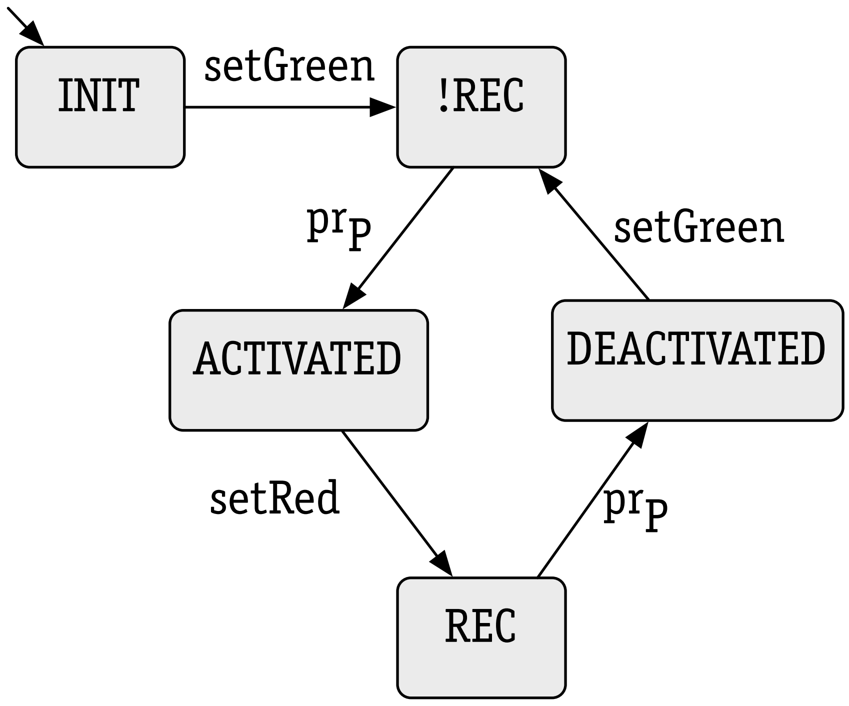
In general, there can be more than one initial state. This allows you to describe overapproximations -- it's another form of non-determinism, which I mentioned before -- allowing an execution to start in any of the initial states. But generally for the machines we'll look at there's just one initial state.

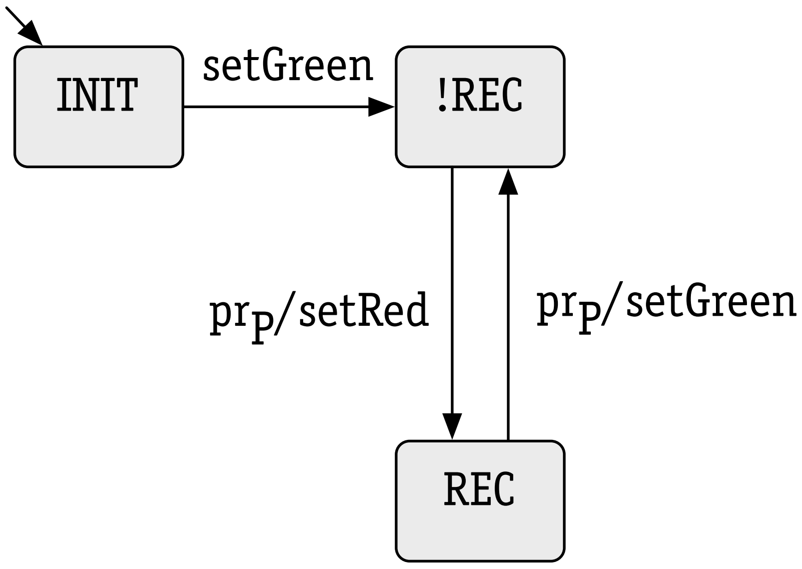
recording

exercise

draw a state machine where

- pressing R key toggles between recording on and off
- and switches indicator light to red (on) and green (off)
- ignore actual recording for now





In doing this exercise, some of you didn't have explicit events for setting the indicator light green and red, and just labeled the states. That's not so good, because as we noted before, the states don't really mean anything: it's only the traces that are observable.

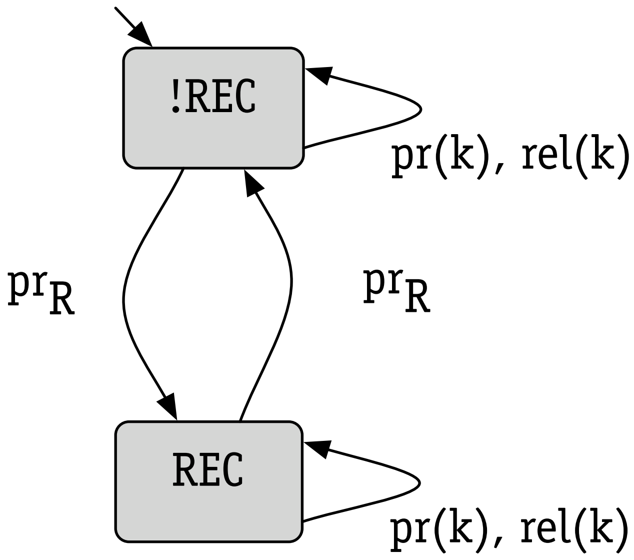
adding recording

design strategy

- add a <u>parallel</u> machine that maintains additional state

new events and states

- events
  - pr_R: keyboard driver reports press of R key
  - pr(k): keyboard driver reports press of any key except for R
- states
  - !REC: recording off
  - REC: recording on

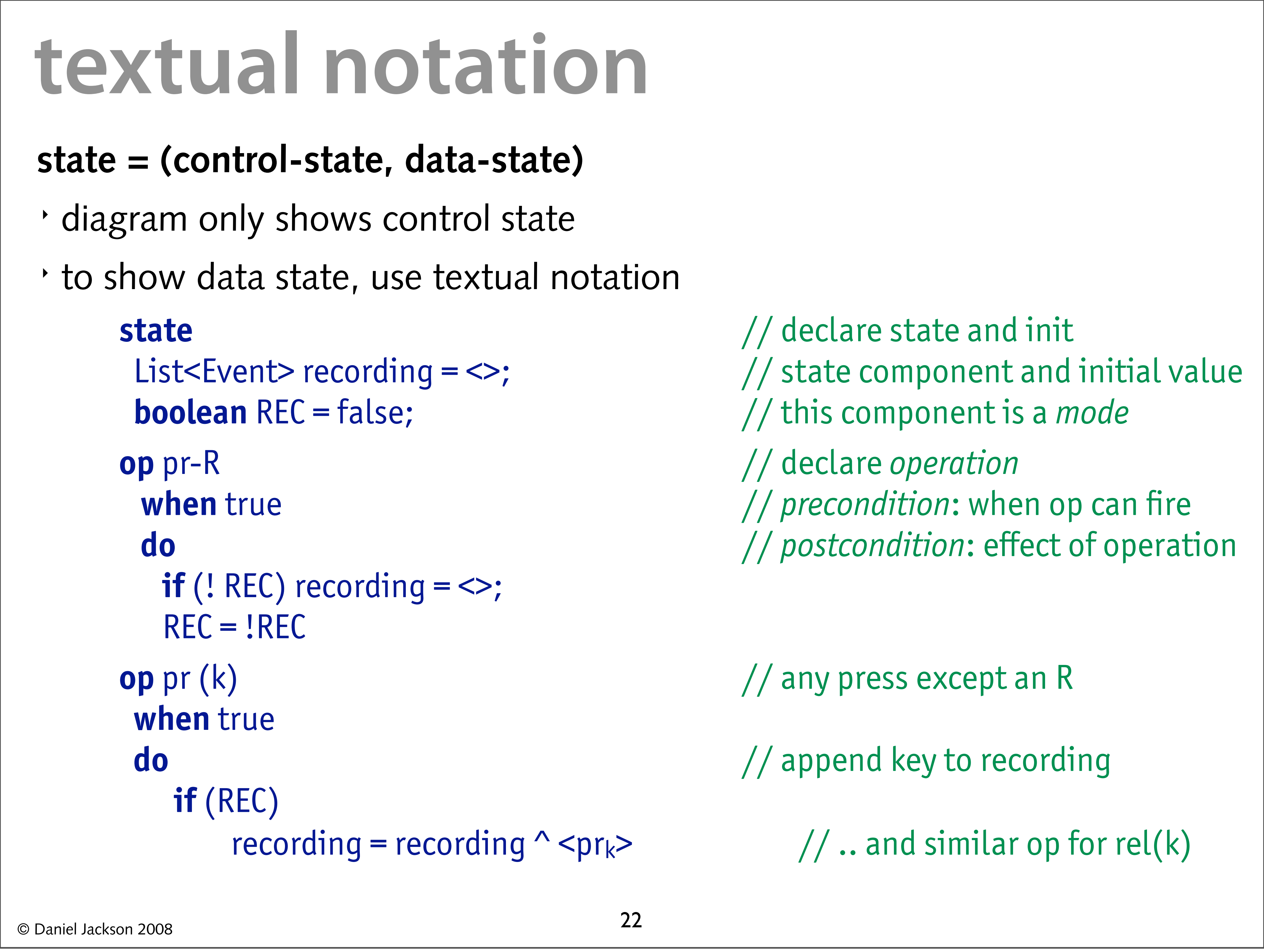


design question

- do we need to worry about key repeats?

You might think that this machine says nothing -- and you'd be almost right. All it does is parse a sequence of keyboard events, deciding whether we're in the recording mode or not after a given sequence. As we noted before, only traces matter and not states. But we'll actually use these states to determine outputs, after elaborating them a bit.

There's no need to worry about key repeats in this case, as we don't expect the user to hold the R key down. If we were worried about them, we could filter them out the way we filtered them out for other keys.



textual notation

state = (control-state, data-state)

- diagram only shows control state
- to show data state, use textual notation

```
state                                        // declare state and init
  List<Event> recording = <>;                // state component and initial value
  boolean REC = false;                       // this component is a mode
op pr-R                                      // declare operation
   when true                                 // precondition: when op can fire
   do                                        // postcondition: effect of operation
     if (! REC) recording = <>;
     REC = !REC
op pr (k)                                    // any press except an R
  when true
  do                                         // append key to recording
     if (REC)
          recording = recording ^ <pr_k>         // .. and similar op for rel(k)
```

Here’s the elaboration. We want to say what actually gets recorded, and we can’t do that in a diagram. This is a textual state machine. It has exactly the same semantics as a diagram, but the states and transitions are now implicit. The set of states is declared by giving state components, and each state is then a combination of values of the components. The transitions are declared by giving operations, each of which defines a set of transitions. For example, the operation pr(k) defines the set of all key press events, where k is the key being pressed. Note that there’s a subtle difference in how I use the terms pr(k) and pr_k. The term pr(k) is a generic press event matching any key k; the term pr_k is a press event for a particular key k. The operation is like a method: you supply the argument k.

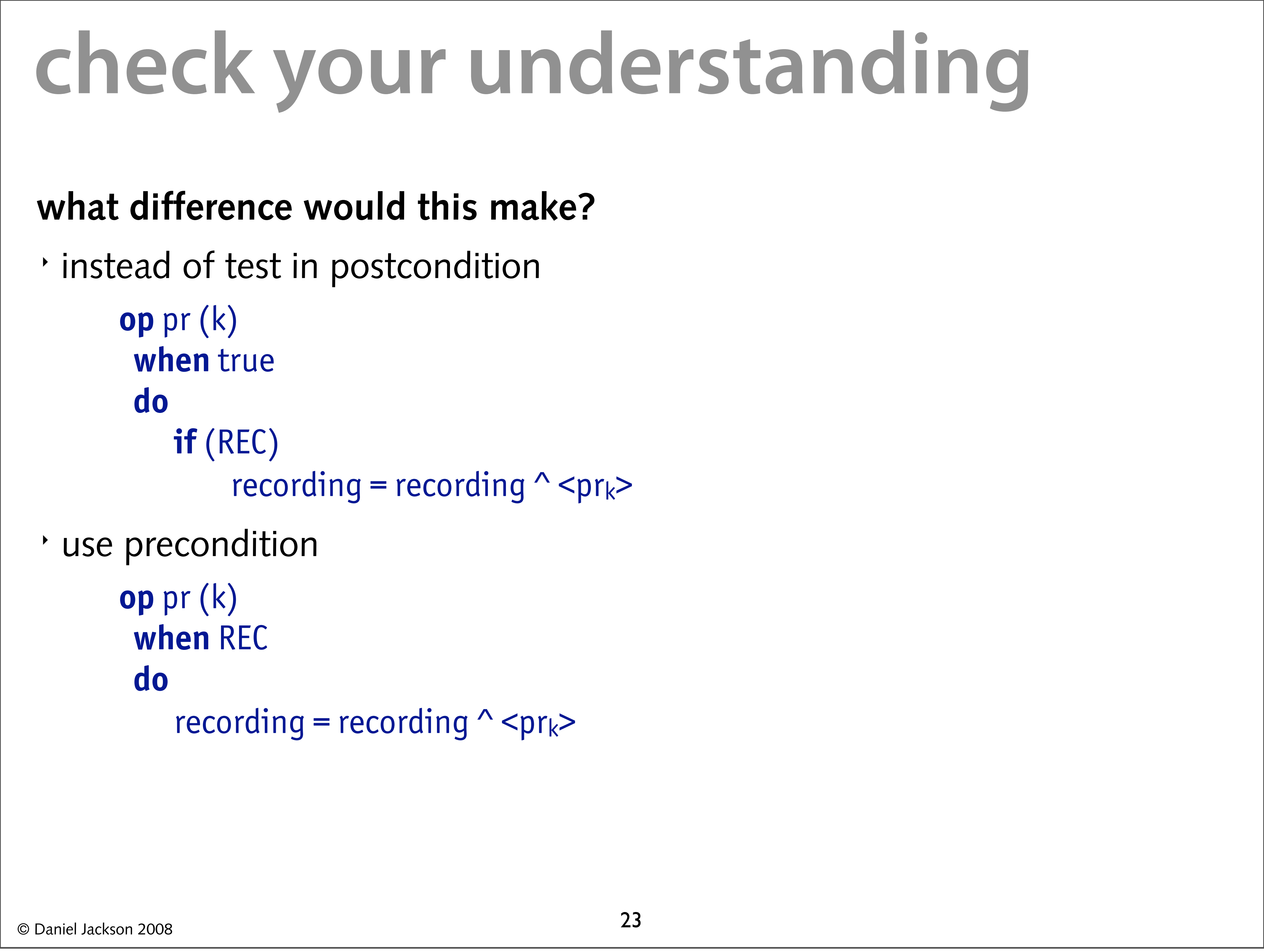
The ! is boolean negation, so REC = !REC just flips the boolean value of REC. This would be bad Java style, btw, to name a variable in all caps. I’m doing it here so that it matches the name of the state in the diagram. When the state machine has one particular state component that takes one of a small set of values, such as REC, it is often called a “mode”. You can think of all the states of the machine being partitioned into the modes: within the mode REC, there are different states corresponding to the different values of the state component recording.

The statement “recording = recording ^ <pr_k>” means that the event of key k being pressed is appended onto the sequence called recording. The ^ is concatenation of sequences, and <e> is the singleton sequence containing just e. This is standard mathematical notation which unfortunately you can’t use in Java.

Note that each operation has two parts, a precondition that says when the operation can occur, and a postcondition describing the way the state is updated. It’s important to understand that although this seems much more complicated than the state machine diagram, the mathematical structure underneath is the same. So the operation pr–R, for example implicitly defines transitions like

(!REC, <e>), pr–R, (REC, <>)
(REC, <e>), pr–R, (!REC, <e>)

where each state has two components. There are just many more of these than for the diagram, because there’s a transition tuple (REC, s) for example for every recording sequence s.



check your understanding

what difference would this make?

- instead of test in postcondition

```
op pr (k)
 when true
 do
    if (REC)
         recording = recording ^ <$pr_k$>
```

- use precondition

```
op pr (k)
 when REC
 do
    recording = recording ^ <$pr_k$>
```

In our example, we didn't actually use the precondition, because every operation was allowed in every state. Here's an example of using it, although it's not what we want in this case. Suppose we put the test from the postcondition into the precondition. What this now says is not that we only update the recording sequence when in mode REC but that the pr-R event can only occur in mode REC, which is a very different thing!

analyzing recording

consider traces

- how about this trace?

  <pr(1), pr_R, rel(1), pr_R>

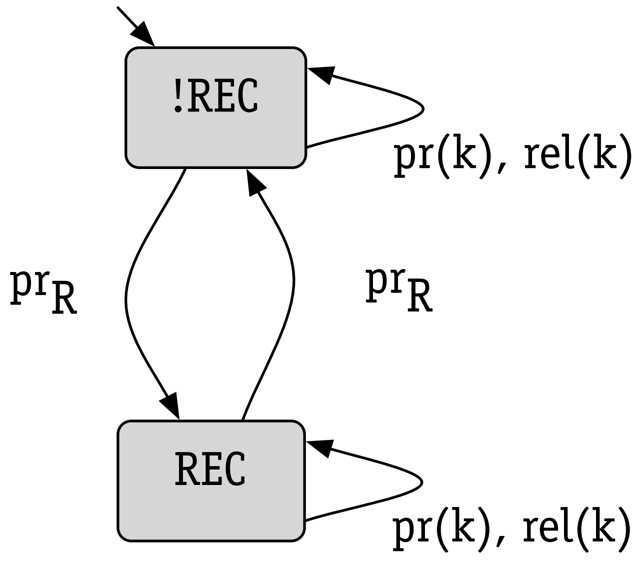
- very strange behavior: recording starts with key release!

two simple options

- filter out initial key releases
- or just allow it (which is what we'll do)

or perhaps

- regard as evidence of larger issue
- maybe recording should be of musical structure and not of user events?



Back to our simple machine. Note that something funny can happen: you can begin a recording with a key being released! There are many ways we could deal with this. We could filter out pr–R events when any other key is depressed. We could do something more complicated and somehow handle the unmatched release events, either by generating press events to match or by dropping them. What I actually did in my implementation, which you'll see in the next lecture, is just to ignore this issue because I wanted to keep things simple.

But actually this little detail is a symptom of a much bigger problem. What's really going on is that we shouldn't be recording press and release events at all. We should be recording musical entities, such as notes of a particular duration and pitch. This little problem here is just the tip of the iceberg. If we go ahead and record only press and release events, we won't be able to associate different instruments with different notes: recording with one instrument and then playing back and mixing with notes played on another instrument. This is a typical occurrence in a software development: a small glitch reveals that actually the entire abstraction is suspect. It's one reason that thinking carefully about the behavior before you start building is important, because it's much easier to make big changes at this point than when we already have a pile of code written.

playback

playback

let's add a playback mode

- use P key for playing back

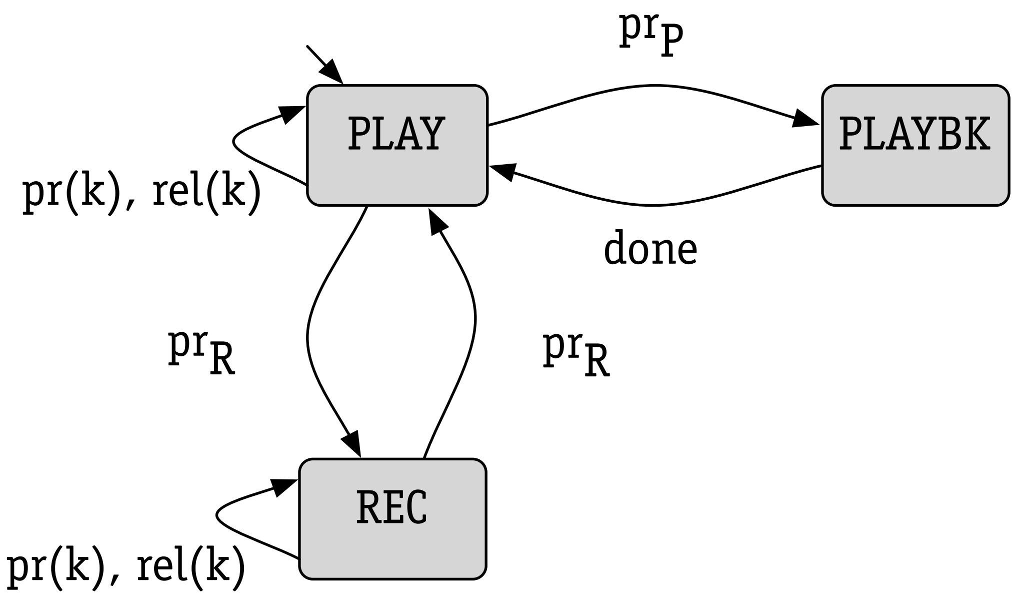
  pr_P: keyboard driver reports P key pressed

- add an event to signal end of playback

  done: output event to signal end of playback

first design option

- playback only enabled when recording is off
- no key presses during playback



This is the most simple notion of playback – it occurs in a mode in which no recording or manual playing is allowed. I'm ignoring for now the question of what exactly is played back; we'll come to that soon.

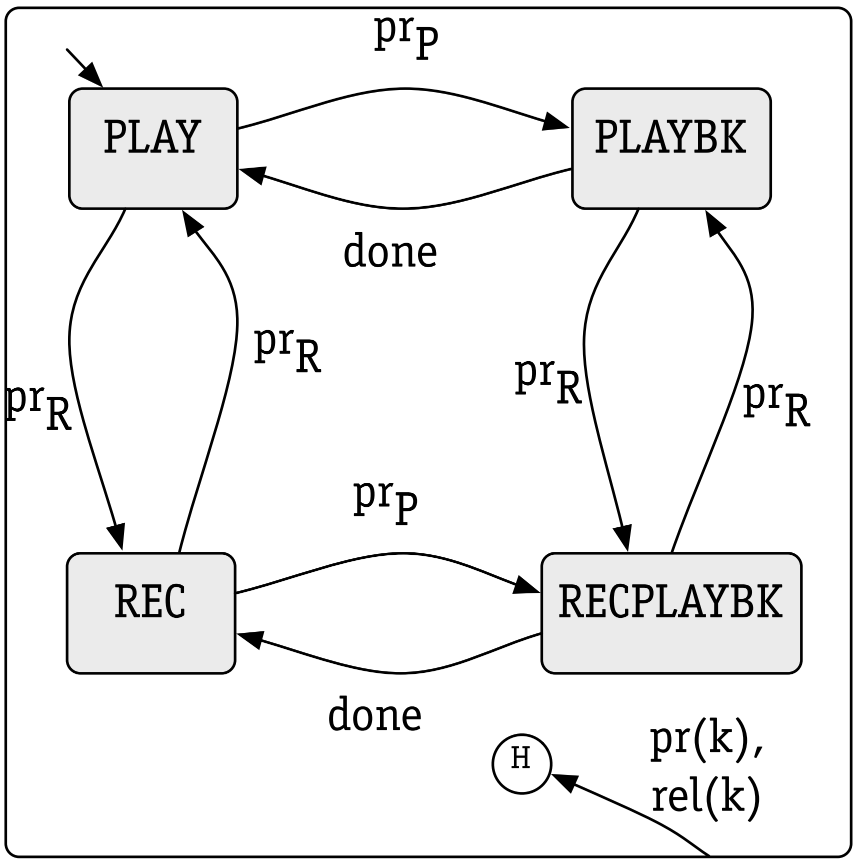
record during playback

can you record during playback?

- very useful, but much trickier

what does playback involve?

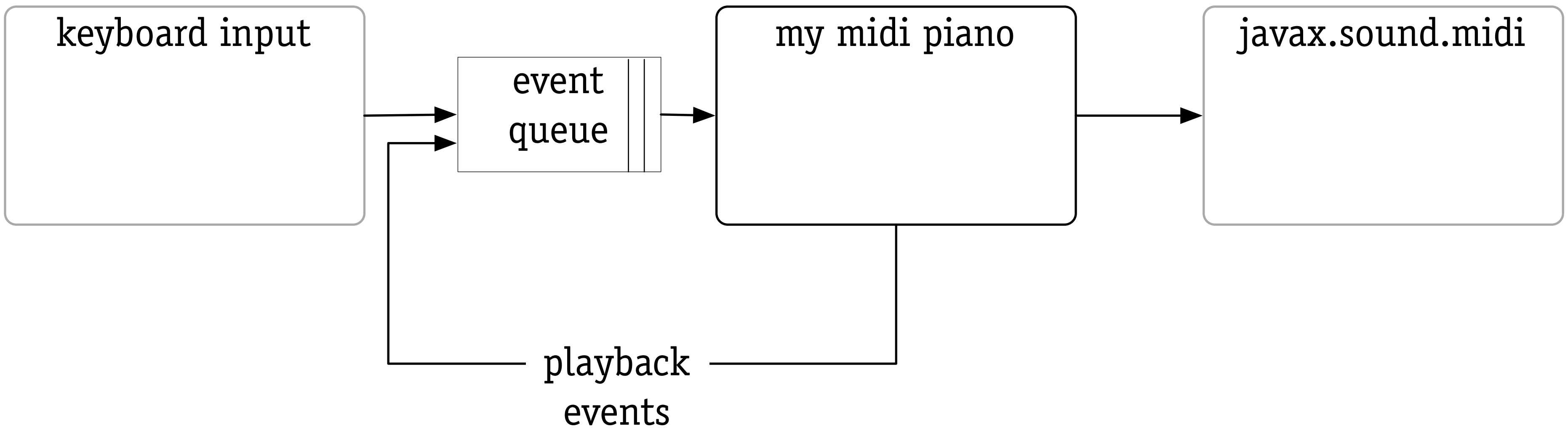
- generated playback events merged with user keyboard events?



This is the more sophisticated and useful notion of playback, with recording and playback interleaved in any way we please. I’m using a statechart shorthand. The arc coming from the outer box labeled “pr(k), rel(k)” says that you can have a transition on a press or release of any key from any of the states inside the box, and the destination of the transition is whatever state you were in before. (H) means history and a transition to that special marker means a transition to the last state that the machine was in inside the larger superstate box.

This still doesn’t resolve the question of what is actually played back. Onto that next...

an asychronous solution



idea

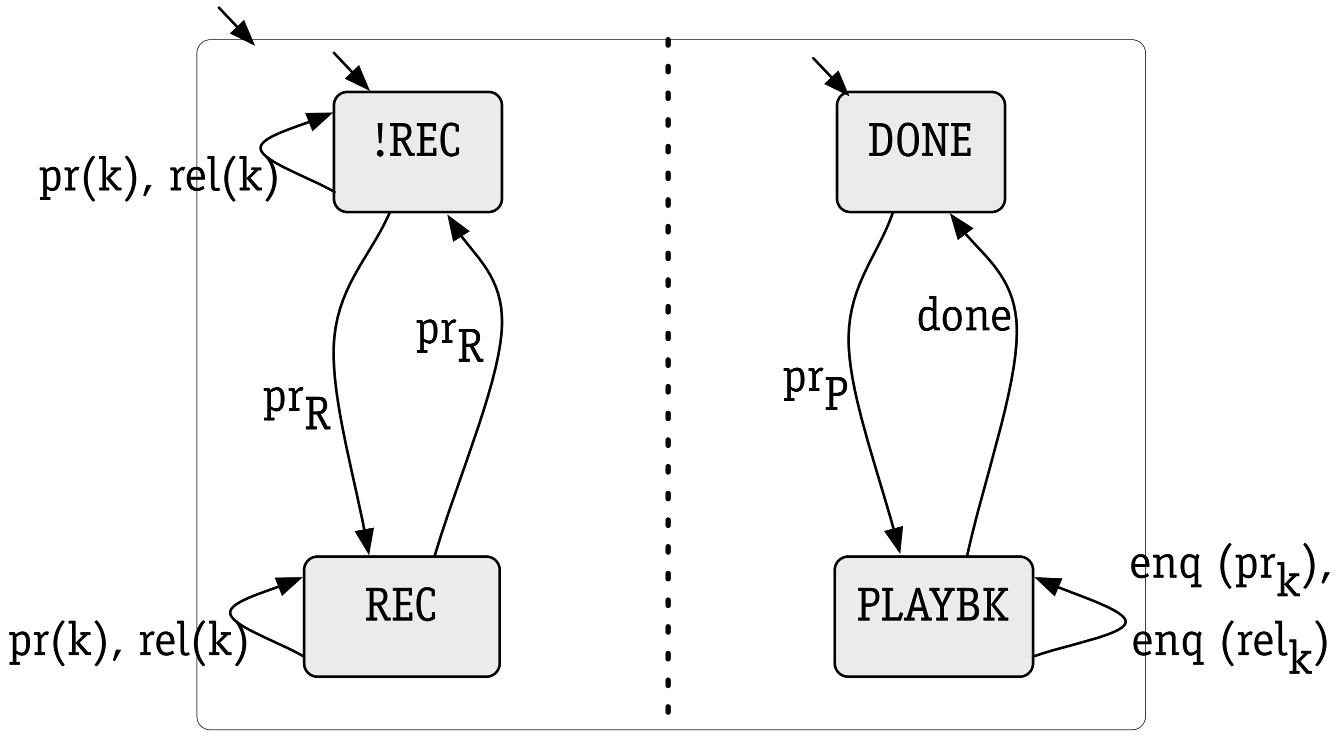
- playback generates press and release events
- merge these events on a queue with incoming keyboard events
- midi piano sees just one stream of events

Here's perhaps the simplest way to handle playback. When the playback key is pressed, all the events in the recording are added to an input queue, along with whatever events are being generated by the keyboard driver. They can be interleaved in an arbitrary way so long as none of the events are delayed too long. As noted above, a better abstraction would be to record and playback musical notes; in that case, we'd have to split the piano module into two parts, and put the queue between the two. This isn't really any more complicated conceptually, but it does require inventing a whole new set of events, and I wanted to avoid that complication. There is, btw, a tricky issue to do with how the events are timed, which I've ignored. A simple solution is to timestamp each press or release with the delay since the last press or release, and have a process that delays putting the next playback event into the queue by that amount.

the playback machine

another over-approximation

- which enq events are generated will depend on what was recorded
- (that is, reads recording state component)



This diagram describes the enqueuing of the events; it shows that during playback any sequence of press and release events can be enqueued. We're abstracting away the details of which events get enqueued, but it's pretty straightforward: it's just the events in the recording sequence that we specified textually.

more on recording state

what would happen if

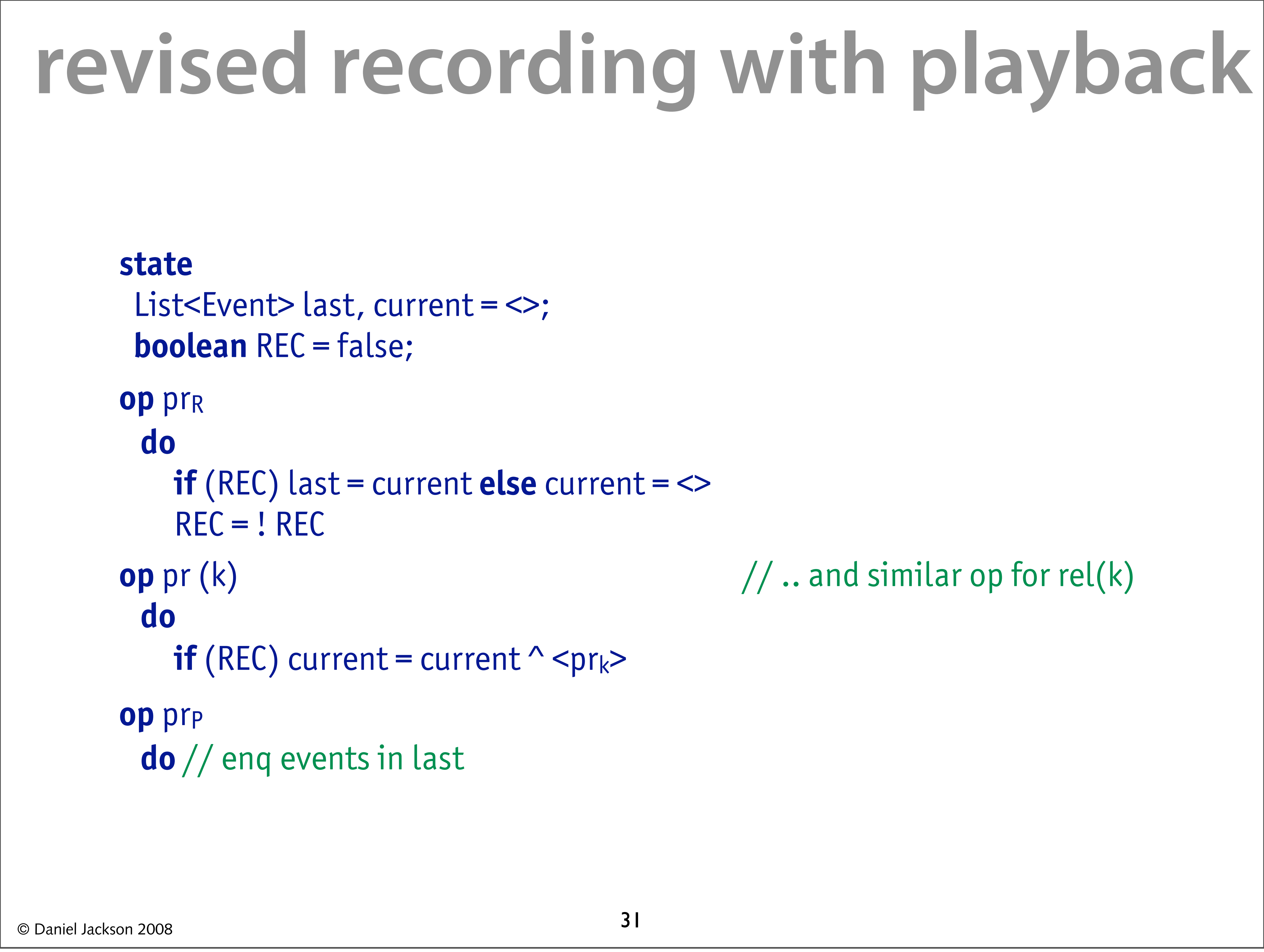
- playback is pressed during record
- generated events get played
- and added to recording
- so they get played back again...

oops!

- keep two recordings

  current: holds events being recorded

  last: holds events being played back
- show how these are updated in textual syntax

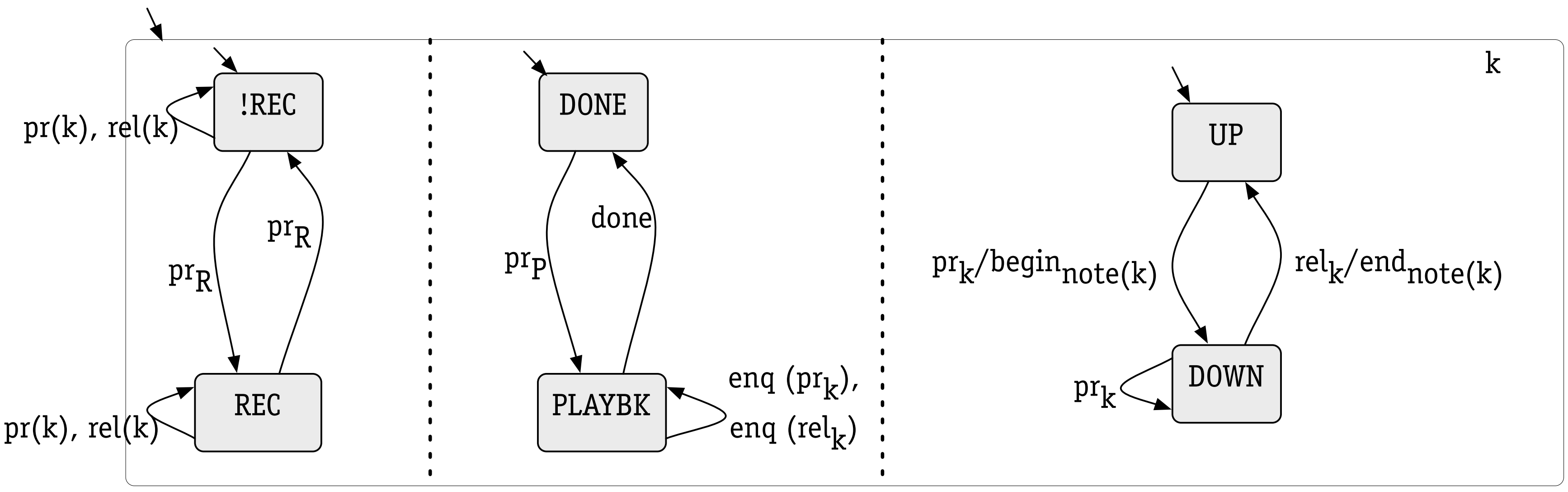


revised recording with playback

```
state
  List<Event> last, current = <>;
  boolean REC = false;
op $pr_R$
  do
    if (REC) last = current else current = <>
    REC = ! REC
op pr (k)                                          // .. and similar op for rel(k)
  do
    if (REC) current = current ^ <$pr_k$>
op $pr_P$
  do // enq events in last
```

Here’s the textual description showing how the two lists are handled. I omitted the preconditions, which means that they are by default true. This description doesn’t actually handle how the events are enqueued.

final state machine



```
state
 List<Event> last, current = <>;
 boolean REC = false;

op $pr_R$
  do if (REC) last = current else current = <>; REC = ! REC

op pr (k)
  do if (REC) current = current ^ <$pr_k$>

op rel (k)
  do if (REC) current = current ^ <$rel_k$>

op $pr_P$
  do // enq events in last
```

This is a summary of what we've arrived at in our state machine behavior design, and it will be the starting point for our design of the mechanism. You can think of the textual part as describing another component state machine that is in parallel with all the others. The first two diagrammatic component machines just summarize this machine from the perspectives of the recording and playback keys.

summary

principles

description before invention

- before you invent something, understand what's there
- especially important for your design's environment
- eg, keyboard driver and midi API

behavior before mechanism, or 'what before how'

- design the behavior of a system first
- easier to design the mechanism when you know what the behavior is
- and easier to analyze the behavior when you can ignore mechanism
- eg, state machines before Java classes

get the details right

- details matter for quality, and can hide major structural flaws
- eg, details of playback during record determine gross structure

addendum

modeling advice

grounding in reality

- select events first and thoughtfully
- give explicit **designations** saying what they mean
- must be atomic and recognizable

bad smells

- events not designated, event classes not really disjoint
- no initial state marked
- states or transitions without labels

a state machine is not a control flow graph

- no behavior inside states
- no ‘decision edges’